my_tcg

Sommaire

Détails administratifs

- Le projet est à réaliser seul
- Les sources devront être rendues avec Blih
- Le nom du rendu est : « my_tcg »
- La stabilité du code devra être garantie par un maximum de tests unitaires

En cas de non-respect d'une de ces règles le projet sera noté 0.

A partir de maintenant, des tests unitaires sont attendus pour chacun de vos projets. Cela sera considéré en soutenance.

Introduction

Ce sujet sera l'occasion d'aborder une notion très importante dans le développement d'application moderne, les tests unitaires. Pour ce faire vous allez développer (la partie logique métier) un jeu de type « Collectible Card Game ». Les tests unitaires permettront de garantir la qualité du code.

Restrictions

Toutes les technologies (JS) sont autorisées, toutefois, si certains outils/technologies sont indiquées au fil du sujet, vous devez les utiliser.

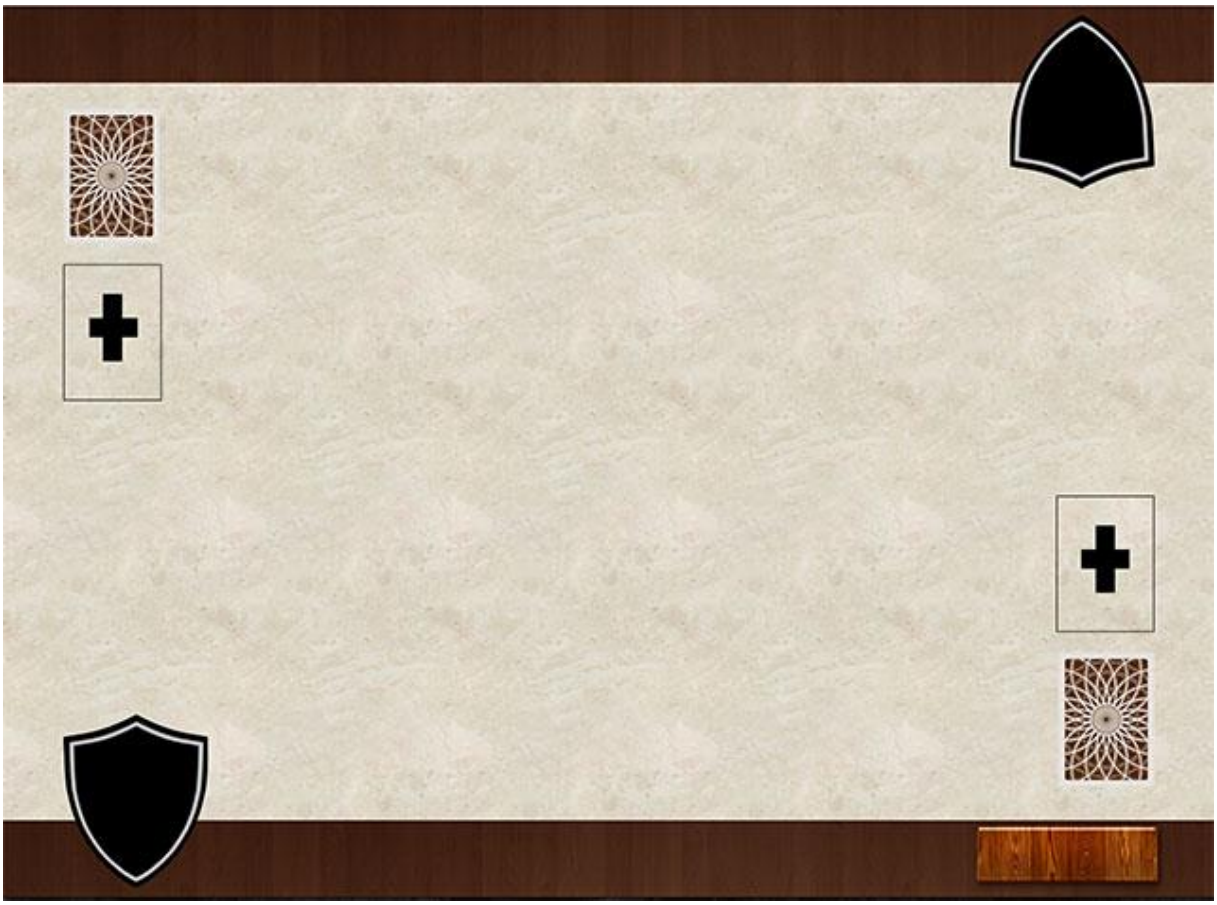
Projet

Première Partie : Installation

Souhaitant vous concentrer sur la conception des mécaniques de votre jeu, vous avez délégué la partie visuelle à un tiers développeur. Par ailleurs, et comme vous avez pu le constater les technologies web évoluent très vite. Vous décidez donc que le projet sera codé en EcmaScript 6 (aussi appelé es2015). Pour ce faire le développeur tiers qui travaille avec vous, vous parle des outils suivants : **gulp, babel, et browserify. Vous irez donc vous renseigner sur le rôle et l'utilisation de chacun**.

Une fois qu'il a fini son travail, le développeur en charge de la partie visuelle vous a livré un pack de scripts (http://lyon-2017.wac.epitech.eu/my_ccg.zip). Mais une fois décompressé, en ouvrant le script index.html (dans le dossier public), vous n'obtenez qu'une page blanche… Vous êtes pourtant convaincu de la fiabilité de la personne qui vous a livré le code. Vous avez donc sans doute oublié de faire quelque chose pour que l'interface s'affiche (il n'y a aucune ligne de code à toucher à cette étape…).

Voici ce que doit afficher la page index.html une fois l'installation du projet terminée :



Deuxième Partie : Mes premiers tests unitaires

L'une des pires choses qui puisse arriver à un développeur (toutes technologies confondues), c'est de provoquer des régressions (de nouveau bugs dans une partie d'application qui fonctionnait jusqu'alors). Il faut refaire (débugger) un travail que l'on a déjà fait, c'est ennuyeux et parfois le code à débuguer n'est plus tout jeune (on ne s'en souviens plus) ou pire encore, on est pas la personne à l'origine du code en question.

Pour se prémunir au maximum de ce genre de complications, et pour simplifier le débogage quand c'est inévitable, les développeurs ont inventé les tests unitaires.

Renseignez-vous sur la philosophie des tests unitaires (pourquoi unitaires ? qu'est-ce que cela implique ?).

Pour vous avancer, dans le début de projet qui vous a été livré, l'environnement de test est déjà prêt (nous utiliserons le framework de test « Jasmine »). Vous devez trouver la commande (à taper dans la console) qui exécute les tests. Vous devriez obtenir quelque chose ressemblant à ceci :

```
> my_hearthstone@0.0.1 test /Users/freakdev/DEV/my_hearthstone/my_hearthstone
> babel-node --presets es2015 spec/run.js

Spec started

Executed 0 of 0 specs SUCCESS in 0.002 sec.
freakdev@FreakBook:~/DEV/my_hearthstone/my_hearthstone (student-pack %) $
```

Le framework « jasmine » promeut la méthode « Behavior Driven Development » (ou « BDD »), qui consiste à écrire les tests au fur et à mesure que l'on décrit le comportement souhaité de l'application (avant d'avoir écrit du code applicatif à proprement parler). Vous irez vous renseigner sur l'intérêt de cette méthode.

Conformément, à la philosophie de la méthode BDD, **nous allons commencer par écrire des tests** en suivant les spécifications suivantes :

- Une seule classe par fichier
- Le nom du fichier sera écrit en « camelCase » et doit avoir le même nom que la classe qu'il contient (ex : pour la classe EventManager, le fichier s'appelle eventManager.js)
- Les classes que vous allez créer dans cette partie du sujet concernent toutes la logique métier de votre application. Elles devront donc toutes être placées dans le dossier « src/models/ »

Classe Deck

Permet de gérer les opérations courantes sur un paquet de carte.

Doit implementer les methodes : shuffle, draw, insertAt, getCardsCount.

Doit avoir la propriété : (array) cards.

Détails des methodes :

constructor(object config)

Accepte un objet en premier argument. Cet objet doit avoir une propriété cards contenant un tableau (de cartes) qui constituerons les cartes du paquet.

mixed draw()

Supprime et retourne la première carte du paquet, ou false s'il n'y a plus de carte dans le paquet.

void shuffle()

Doit mélanger les cartes du paquet, retourne true si le paquet a été mélangé. False sinon.

bool insertAt(mixed card, [int position])

Doit permettre d'ajouter une carte (premier paramètre) dans le paquet à une position donnée en deuxième paramètre.

int getCardsCount()

Retourne le nombre de cartes actuellement dans le deck.

Faites exécuter les tests. Si vous n'avez pas d'erreurs (erreur de syntaxe, mauvaise utilisation du framework, etc…), le rapport d'exécution (ce qui s'affiche) devrait vous dire que les tests « FAIL ».

Finissez d'implémenter la classe Deck (dans le fichier src/models/deck). Une fois que vos tests « pass » (indique « SUCCESS »), et tant qu'il en sera ainsi, cela signifiera que vous n'avez pas à vous souciez de la classe Deck, elle fait ce qu'on attend d'elle (donc aussi les différentes parties de son code). Pour preuve, si vous retournez et exécutez le nouveau code dans un navigateur, en cliquant sur l'un des decks, vous devriez être en mesure de faire piocher une carte à l'un des joueurs.

Nous allons donc passer à la suite, **en commençant d'abord par écrire les tests correspondants (aux spécifications) des différentes classes…**

Note :

- Pour la suite vous pouvez, au choix, écrire tous les tests (de toutes les classes), puis implémenter (toutes) les classes, ou implémenter les classes au fur et à mesure (une fois que les tests correspondant ont été écrits). Quoi qu'il en soit, relier « l'intelligence de jeu » (vos

classes) et l'affichage (le code qui vous ai fourni) comme pour le Deck, n'est prévu que dans la 3e partie du sujet.
- Certaines classes sont déjà créées, il s'agit alors de les modifier / compléter selon les spécifications.

Comme dans toute conception en POO, les objets auront un rôle précis et délimité. Nous allons donc faire des classes pour chaque « aspect » du jeu.

Classe Hand

Permet de gérer la main d'un joueur.

Doit implémenter les méthodes suivantes : addCard, removeCard, getAllCards, getCardsCount.

Doit avoir la propriété suivante : (int) limit.

Détails des méthodes :

constructor(object config)

Accepte un objet en premier argument. Cet objet doit avoir une propriété cards contenant un tableau (de carte) qui constituerons les cartes de la main

On pourra préciser une limite maximum de carte qu'il est possible d'avoir en main. Pour ce faire, il faudra préciser une propriété « limit » dans l'objet passé en paramètre au constructeur.

Si cette limite n'est pas précisée, la valeur par défaut de cette limite sera 7.

bool addCard(object card)

Ajoute une carte (qui est passé en paramètre) à la fin de la liste des cartes déjà dans la main. Si tout s'est bien passé, la fonction retourne true, sinon false.

Aucune carte ne doit être ajoutée si la limite est déjà atteinte, la méthode doit retourner false.

bool removeCard(int position)

Retire de la main la carte positionnée à l'index passé en paramètre. Retourne la carte s'il n'y a pas eu de souci, false sinon.

array getAllCards()

Retourne un tableau contenant toutes les cartes de la main.

int getCardsCount()

Retourne le nombre de carte actuellement dans la main.

Class Board

Permet de gérer un board (l'ensemble des cartes mises en jeu).

Doit étendre la classe Hand.

Class Cemetary

Permet de gérer un cimetière.

Doit étendre la classe Deck.

Class Pawn

Classe abstraite qui sert de base a tout ce qui pourra combattre.

Doit étendre la classe EventManager.

Doit implémenter les methodes suivantes : getLife, getStrength, getDef, attack, recieveAttack.

Détails des méthodes :

constructor(int life, int strength, int def)

Accepte 3 entiers en paramètre, le premier étant le niveau de vie initial, le second la force, le dernier la défense.

getters : int getLife(), int getStrength(), int getDef()

Permet de lire les valeurs des différents attributs de l'objet.

bool attack(Pawn target)

Accepte en paramètre un objet (ci-après désigné par « target ») qui sera lui aussi une instance de Pawn (ou d'une classe dérivée).

Attaquer consiste à invoquer la méthode recieveAttack de l'objet target avec en seul paramètre une instance de l'attaquant (l'objet attaquant lui-même).

bool recieveAttack(Pawn opponent, bool strikeBack = false)

Cette méthode doit décrémenter la vie de l'objet courant de la valeur de la force de l'attaquant (passé en premier paramètre), puis effectuer une contre-attaque (appeler la méthode recieveAttack de l'attaquant avec true en second paramètre).

A l'exécution de cette méthode si le 2nd paramètre est présent et qu'il vaut « true » (il s'agit d'une contre-attaque), on décrémentera la vie de la valeur de la défense de l'(contre-)attaquant, non de la valeur de la force, et on ne fera pas de contre-contre-attaque.

Class Player

Permet la gestion d'un joueur.

Doit étendre la classe Pawn.

Doit implémenter les méthodes suivantes : shuffle, draw, playCard, discard, attack.

Doit avoir les propriétés suivantes : (Deck) deck, (Board) board, (Hand) hand, (Cemetary) cemetary

Détails des méthodes :

constructor(object config)

Accepte un objet de configuration en paramètre. Cet objet, pour initialiser le deck du joueur, doit avoir une propriété deck qui sera soit un tableau, soit une instance de la classe Deck.

bool shuffle(string deck = "deck")

Accepte en paramètre une chaine de caractère servant à désigner quel paquet mélanger.

Les valeurs possibles sont « deck » et « cemetary ».

Si un paquet a été mélangé, la méthode retourne true, dans tous les autres cas, elle retourne false.

bool draw()

Doit piocher la première carte du deck du joueur et l'ajouter à la main du joueur. Retourne la carte piochée si toutes les opérations se sont bien passées, false sinon.

bool playCard(int position)

Retire la carte, à la position indiquée en paramètre, de la main du joueur et l'ajoute à son board. Retourne true en cas de succès, false sinon.

bool discard(int position)

Retire la carte, à la position indiquée en paramètre, de la main du jour et l'ajoute à son cimetière. Retourne true Si toutes les opérations se sont bien passées, false sinon.

bool attack(int position, Pawn target)

Doit déclencher une attaque avec la carte du board désignée par le premier paramètre, et cibler l'adversaire passée en second paramètre.

Class Game

Permet l'encapsulation du jeu dans une seule instance.

Doit étendre la class EventManager.

Doit implémenter les méthodes suivantes : proxy, getTurn, changeTurn.

Détails des méthodes :

constructor(object config)

Accepte en paramètre un objet avec 2 propriétés « up » et « down » chacune d'elle devra représenter un des adversaires.

string getTurn()

Retourne une chaine qui indique le tour en cours (« up » ou « down »)

string changeTurn()

Change le tour courant et retourne le nouveau tour (« up » ou « down »)

bool proxy(string side, string action, mixed payload)

Exécute la méthode indiquée par le paramètre « action » pour le joueur désigné par le paramètre « side ». Le « payload » sera passé en paramètre à la méthode appelée.

Si tout s'est bien passé la méthode proxy retourne le résultat de la méthode invoquée, false sinon.

Troisième partie : Let's play

Vous avez désormais toutes les cartes en main pour implémenter un jeu de type « Collectible Card Game ».

Un peu d'explication sur le fonctionnement de la partie graphique de l'application pourra s'avérer utile. De la même façon que nous avons découpé la logique de jeu de façon à encapsuler les unes dans les autres les différentes parties de l'application, le code qui gère l'interface graphique est, de la même façon, découpé selon les différents « composant » et encapsulé les un dans les autres.

Les écouteurs d'événements DOM (clic par exemple) sont ajoutés au niveau du composant, puis, s'il y a lieu les « événement » sont remontés au composant parent. Les événements peuvent être remonté jusqu'à la racine des composants (l'arène). C'est ce dernier (et lui uniquement) qui devra

communiquer avec le modèle, puis éventuellement (en fonction des retours du modèle) lancer à son tour un « événement » qui sera écouté par les composants enfants.

La partie graphique étant bien séparée de la logique de jeu, les « points de contact » étant identifiés et limité (seul l'arène doit communiquer avec le modèle), la logique de jeu peut être aisément portée coté serveur pour permettre de jouer à 2. Les tests unitaires nous permettront de nous assurer qu'une fois le code porté côté serveur, la logique n'aura pas changé.

Pas de restriction quant aux règles du jeu, vous pouvez implémenter les règles qui vous plaisent, attention toutefois à conserver les précédentes (ce sont elles qui comptent le plus dans la notation).